National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

# A Proof Procedure for the Preferential Semantics

BY

Zhong Li

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Masters of Science.

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Fall 1993

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

*Your file Votre référence*

*Our file Notre référence*

**The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.**

**L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.**

**The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.**

**L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.**

ISBN 0-315-88213-1

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Zhong Li

TITLE OF THESIS: A Proof Procedure for the Preferential Semantics

DEGREE: Masters of Science

YEAR THIS DEGREE GRANTED: 1993

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

(Signed) . . . . . .

Zhong Li
140 Kiniski Crescent
Edmonton, Alberta
Canada T6L 5A9

Date: July 14, 1993

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled A PROOF PROCEDURE FOR THE PREFERENTIAL SEMANTICS submitted by Zhong Li in partial fulfillment of the requirements for the degree of Masters of Science.

Jia-Huai You

Dr. Jia-Huai You (Supervisor)

Bernard Linsky

Dr. Bernard Linsky (External)

Peter v. Beek

Dr. Peter van Beek (Examiner)

Dr. Piotr Rudnicki (Chair)

Date: June 17, 1993

Dedicated to my parents for giving my life
and
my wife and my daughter for enriching my life

# Abstract

The recent development of various declarative semantics and their proof procedures for normal logic programs reflects the need to have better understanding of normal logic programs both semantically and operationally. How to treat negation in a logic program becomes a difficult topic which has received a lot of attention in recent years. Many different semantics have been proposed for normal logic programs. Dung's preferential semantics has shown some advantages over others.

To compute the intersection of all the extensions for a given theory has been a difficult topic in nonmonotonic reasoning and logic programming. We propose a new proof procedure, also the first one according to our knowledge, for the preferential semantics from the skeptical reasoning point of view. This proof procedure uses both top-down and bottom-up methods to make reasoning. The key idea of our method is to define two different proof trees, called *Primary Proof Tree* and *Assumption Proof Tree* respectively. By checking the consistency of those two proof trees' leaves we can answer a query correctly for a given logic program. We show the soundness of our method with respect to the preferential semantics and present some analyses of completeness and computational complexity.

# Acknowledgements

I would like to express my deepest thank to my supervisor, Dr. Jia-Huai You, for his continued guidance and encouragement. His never-ending intellectual, moral, ideological, and financial support makes this thesis possible. His insightful comments in every step of my research have helped me reach my potential. Over the years, his generous helps make me from a student to a researcher. Moreover, I have received a lot of help from him in my personal life.

I am grateful to the members of my examining committee, Dr. Peter van Beek and Dr. Bernard Linsky for their time and effort in reviewing my research work and making thoughtful comments on my thesis. Also thank you, Dr. Piotr Rudnicki, for chairing the committee.

Thanks to Pablo Hadjinian, Aditya Ghose, Allen Sharpe, and Suryanil Ghosh for their valuable comments and discussions throughout my research. Aditya deserves a special thank for organizing AI group weekly meeting which has brought me a lot inspirations.

# Contents


**1 Introduction** **1**

1.1 Background . . . . . 1

1.2 Motivation and Our Work . . . . . 5

**2 Negation in Logic Programming** **7**

2.1 Preliminaries . . . . . 8

2.2 Resolution . . . . . 12

2.3 Fixpoints . . . . . 13

2.4 Negation as Failure . . . . . 15

2.5 Completions of Programs . . . . . 16

2.6 Stable Models . . . . . 17

2.7 Well-Founded Models . . . . . 20

**3 A New Proof Procedure** **24**

3.1 Preferential Semantics . . . . . 25

3.2 A Proof Procedure for Skeptical Reasoning . . . . . 28

3.3 Examples . . . . . 38

3.4 Implementations . . . . . 45

**4 Some Theoretical Results and Analyses** **48**

4.1 Soundness . . . . . 48

4.2 Completeness and Computational Complexity . . . . . 54

**5 Conclusions** **56**

**Bibliography** **58**

# List of Figures


3.1 The Proof Tree of $c$ . . . . . 30

3.2 Two A-trees for $c$ . . . . . 31

3.3 P-tree for $c$ . . . . . 34

3.4 (a) P-tree of $a$ (b) A-tree of $b$ . . . . . 39

3.5 (a) P-tree $PT_{\leftarrow a}$ (b) A-tree $PT_{\leftarrow b}$ . . . . . 40

3.6 (a) P-tree $PT_{\leftarrow a}$ (b) A-tree $AT_{\leftarrow b}$ . . . . . 41

3.7 P-tree for $a$ . . . . . 42

3.8 P-tree $PT_{\leftarrow c}$ for query $\leftarrow c$ . . . . . 43

3.9 P-tree $PT_{\leftarrow c}$ for query $\leftarrow c$ . . . . . 44

4.1 P-tree for $\leftarrow q$ when $N = 2$ . . . . . 49

4.2 Two A-trees for $\leftarrow l_{1i}$ and $\leftarrow l_{2j}$ respectively . . . . . 49

4.3 A-tree for $AT_{\leftarrow l}$ . . . . . 51

# Chapter 1

# Introduction

In this chapter we give a brief introduction to negation in logic programming and the different approaches which have been proposed in recent years for how to treat negation. We also provide the motivation of our work and outline the main results to be reported in this thesis.

## 1.1 Background

The original idea of designing symbolic logic was to use it as a formalization of human reasoning. As a result, it has long been used in computing science as a specification language for computer programs and as a foundation for database query languages. The conflict between expressive power and efficiency of execution has always been a major problem faced by the designers of programming languages. In the conventional approach, expressive power has consistently been sacrificed in an ad hoc manner for efficiency and has often been confused with the accumulation of "useful" features [JLM86]. The two relevant aspects of predicate logic are the *model theory* and the *proof theory*: model theory corresponds to specification and declarative notions while proof theory corresponds to operational semantics

which provides guidance for implementation. The model theory (or *declarative semantics*) describes a precise meaning of a logic program which is independent of procedural considerations. The proof theory (or *procedural semantics*), on the other hand, is given by providing a procedural mechanism. The behavior of such a mechanism (especially its correctness) is evaluated by comparing its behavior to the *specification* provided by the declarative semantics.

There are no existing "suitable" declarative semantics for logic programs. One of the difficulties in finding "suitable" declarative semantics is that there is no precisely defined set of conditions such that the "suitable" declarative semantics satisfies this set of conditions. In fact, different declarative semantics are usually based on different intuitions. There are clashes: a declarative semantics may be intuitive to some people but may not be intuitive to others, and different declarative semantic treatments may be suitable for different applications. However, it seems that it has been widely accepted that a declarative semantics should at least be reducible to the *perfect model semantics* [Prz88] for (locally) *stratified* logic programs. Several alternative formulations of negation that appear to be equivalent to the stable and the well-founded semantics have been developed (see [Bry89],[Prz89a], and [Gel92]). This tells us that a declarative semantics of a logic program should be determined more by its ***common sense meaning*** than by its purely logic contents. For example, given a list of names registered in a course, we should be able to reach a common sense conclusion that John does not take this course if his name does not appear in this list.

The "intended" semantics (from now on when we say semantics we mean the declarative semantics) must therefore be nonmonotonic, i.e., the conclusions determined by a semantics of a logic program P cannot monotonically increase. For example, after learning that John has been added into the name list, we have to withdraw the previously reached conclusion that John does not take this course. Consequently, the problem of finding a suitable semantics for logic programs can

be viewed as the problem of finding a suitable semantics in a nonmonotonic formalization of the type of reasoning used in logic programs. *Negation* in logic programs has played a very important role in finding "suitable" semantics because the differences of semantics are mostly reflected in the treatments of negations.

As for the procedural semantics, Apt and Van Emden([AE82]) introduced fixpoint techniques to establish various results in an elegant way. For instance the soundness and completeness of SLD resolution with respect to *definite programs*. SLD-resolution stands for SL-resolution for Definite clauses. SL stands for *Linear resolution* with *Selection function* (see [Llo87]). Their methods have become standard since then. However, they met with difficulties when characterizing the finite failure of SLD resolution to prove an atom to be a logical consequence. This led to long, involved proofs and a weak result which is difficult to interpret semantically: the SLD finite failure set for a given logic program $P$ is equal to the complement of $T_P \downarrow \omega$. $T_P$ is a monotonic operator which maps interpretations of $P$ to interpretations of $P$. The descending ordinal powers $T \downarrow \alpha$ of $T_P$ are defined by $T_P \downarrow 0 = B_P$ ($B_P$ is the Herbrand base of $P$), $T_P \downarrow (\alpha + 1) = T_P(T_P \downarrow \alpha)$, $T_P \downarrow \alpha = \bigcap_{\beta<\alpha} T_P \downarrow \beta$ for a limit ordinal. The greatest fixpoint of $T_P$ is equal to $T_P \downarrow \alpha$. The result is difficult to interpret semantically, as the complement of $T_P \downarrow \omega$ is an abstract mathematical object implicitly defined. We only have a purely mathematical characterization of the SLD finite failure set. It is weak in the sense that for some $l$ in Herbrand base $l \notin T \downarrow \omega$ only implies that there exits a finitely failed SLD tree for $l$ while infinite SLD trees for $l$ may also exist.

As Lassez and Maher in [LM84] made it clear that the standard concept of fairness should be introduced in SLD resolution in order to remove a class of infinite computations due to the interpreter's design. The result was startling; most difficulties in the treatment of SLD finite failure were due to the presence of these unnecessary loops. With this modification, all trees for a given goal behave

in the same way: either all of them fail or none of them fails. Furthermore , if some SLD tree for a literal $q$ finitely fails, then all SLD trees for $q$ are finitely failed. Consequently, if we have an implementation of SLD resolution, testing the SLD finite failure simply requires the generation of a single tree, as it is the case when the SLD tree succeeds.

By definition, when we compute partial recursive functions, there are inputs which lead to infinite computations. As Kowalski ([Kow79]) pointed out, in the context of logic programming there is no limit to the amount of infinite branches that can be pruned from the search space (by loop-checking methods etc.), and there could be no way to eliminate all of them. Consequently, we can further and further approximate the *closed world assumption* but never, in general, reach it.

How to treat negation in a logic program has been a topic extensively investigated in recent years. This has also resulted in much research in databases and nonmonotonic reasoning. Clark ([Cla78]) provided a formal framework to study negation as failure via the notion of *complete logic programs*, and proved the soundness of negation as failure. This turned out to be quite important as it provided a better understanding of the negation as failure rule and opened an active area of research.

There are two major approaches to the semantics of negation in logic programming: the *stable semantics* and the *well-founded semantics*. While the stable semantics is based on a subset of two-valued minimal models, the well-founded semantics is defined by a three-valued minimal model by allowing the status of a predicate to be undetermined. The stable semantics and the well-founded semantics capture different features of a given program. Besides these two major approaches, Dung proposed a new semantics, called the *preferential semantics* [Dun91]. This semantics overcomes some of the drawbacks of the stable semantics and the well-founded semantics.

## 1.2 Motivation and Our Work

Motivated by Eshghi and Kowalski's proof procedure ([EK89]) for abduction, Dung proposed a new semantics, called *preferential semantics*. He showed that the new semantics captures, generalizes and unifies the different existing semantic concepts (e.g. the well-founded semantics, the stable semantics) in logic programming. He also proved the soundness of this proof procedure from the *brave reasoning* point of view with respect to the preferential semantics. Unfortunately, it is not complete with respect to the preferential semantics. The idea of this proof procedure is to simulate negation as failure by making negative conditions abducible and by imposing appropriate denials and disjunctions as integrity constraints. This gives an alternative semantics for negation as failure. The abducible extension of logic programming extends negation as failure in three ways:

1. computation can be performed in alternative minimal models
2. positive as well as negative conditions can be made abducible
3. other integrity constraints can also be accommodated

Although the proof procedure is very elegant for answering a query which is in any extension of a given logic program, it does not allow for answering a query which is in all the extensions of a given logic program. We propose a totally different method using both top-down and bottom-up strategies. Based on these strategies, a proof procedure is developed for answering a query with respect to all the extensions of a given logic program. The key idea of our method is to define two proof trees, called the *Primary Proof Tree* and the *Assumption Proof Tree*. By checking the consistency of these two proof trees' leaves we can answer the query correctly. The bottom-up strategy is reflected in the computation of the well-founded set and the top-down strategy is reflected in the computation of

the proof trees. According to our knowledge, our approach is not only new, but is also a first attempt to compute an extension in the sense of skeptical reasoning for the preferential semantics. In this thesis, we will prove the soundness of our proof procedure with respect to the preferential semantics and present some analyses for the issues of completeness and computational complexity for the case of propositional logic programs.

# Chapter 2

# Negation in Logic Programming

It is important to realize that the use of negation indeed increases the expressive power of logic programs. This may sound paradoxical since, as is well known from [Tar77], logic programs without negation have the full power of recursion theory. But the point is that in many situations we compute over a finite domain, and this drastically changes the situation.

The realization of this fact led to extensive studies, started by Clark [Cla78], of the extensions of logic programming incorporating the use of negation. Two major approaches, recognized by most researchers, to the semantics of negation in logic programming are the *stable semantics* and the *well-founded semantics*. A new comer, called *preferential semantics* which is proposed in [Dun91], has recently received much attention.

In this chapter we first introduce some fundamental definitions which are used throughout this thesis. Then, we briefly discuss different approaches towards negation in logic program. These approaches include negation as failure, predicate completion, stable model, and well-founded model. Since our focus is on logic programming semantics and not on a complete survey of nonmonotonic reasoning,

there are some well-known methods, such as the *closed world assumptions* [Rei78], *perfect model semantics* [Prz88], etc., which have not been mentioned here. All the definitions and theorems in this chapter are cited from [Llo87] if not explicitly stated.

## 2.1 Preliminaries

This section defines the syntax of well-formed formulas for first order logic. All the requisite concepts from first order logic are discussed informally in this section. A first order logic has two aspects: *syntax* and *semantics*. The syntactic aspect is concerned with well-formed formulas admitted by the grammar of a formal language. The semantic aspect is concerned with the meanings attached to the well-formed formulas and the symbols they contain.

**Definition 1** *A* `term` *is defined inductively as follows:*

- *A variable is a term.*
- *A constant is a term.*
- *if $f$ is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. A 0-ary function symbol is a* `constant symbol`.

□

**Definition 2** *A* `(well-formed) formula` *is defined inductively as follows:*

- *If $p$ is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a formula (or* `atom`*).*

- *If $p$ and $q$ are formulas, then so are $(\neg p), (p \wedge q), (p \vee q), (p \rightarrow q)$ and $(p \leftrightarrow q)$.*
- *If $p$ is a formula and $x$ is a variable, then $(\forall x\ p)$ and $(\exists x\ p)$ are formulas.*

□

**Definition 3** *The* `first order language` *consists of the set of all formulas which can be constructed from the symbols of a given alphabet. A* `literal` *is either an atom or the negation of an atom. A* `positive literal` *is an atom. A* `negative literal` *is the negation of an atom.*

□

**Definition 4** *A* `closed formula` *is a formula with every variable appearing in the formula either bounded by $\exists$ or bounded by $\forall$.*

□

**Definition 5** *A* `clause` *is a formula of the form*

$$\forall x_1 \ldots \forall x_s (r_1 \vee \ldots \vee r_m)$$

*or*

$$p_1, \ldots, p_k \leftarrow q_1, \ldots, q_n$$

*where each $r_i$ is a literal and $x_1, \ldots, x_s$ are all the variables occurring in $r_1 \vee \ldots \vee r_m$ or in $p_1, \ldots, p_k, q_1, \ldots, q_n$ ($p_i$ and $q_j$ are atoms).*

**Definition 6** *A* `definite clause` *is a clause in the form of*

$$p \leftarrow q_1, \ldots, q_n$$

*which contains precisely one atom $p$ in its consequent. $p$ is called the* `head` *and $q_1, \ldots, q_n$ is called the* `body` *of the program clause. A* `definite program` *consists*

*of a finite set of definite clauses. If $n = 0$, then the clause is simply $p$ and is called a* `fact`.

□

**Definition 7** *A* `definite goal` *is a clause of the form*

$$\leftarrow q_1, \ldots, q_n.$$

*I.e. the head of the clause is empty. Each $q_i$ is called a subgoal of the goal.*

□

**Definition 8** *A* `Horn clause` *is a clause which is either a definite clause or a definite goal, i.e. a clause with at most one positive literal.*

□

**Definition 9** *A* `ground term` *is a term without variables. Similarly, a* `ground atom` *is an atom without variables.*

□

**Definition 10** *An* `interpretation` *consists of some domain of universe over which the variables range, an assignment to each constant of an element of the domain, an assignment to each function symbol, and an assignment to each predicate (i.e., ground atom) over the domain.*

□

An interpretation thus specifies a meaning for a formula. We are particularly interested in interpretations for which the formula expresses a true statement in that interpretation. Such an interpretation is called a *model* of the formula. Normally there is some distinguished interpretation, called the *intended* interpretation, which gives the principal meaning of the symbols.

**Definition 11** *Let $S$ be a set of closed formulas and $F$ be a closed formula of a first order language $L$. We say $F$ is a* `logical consequence` *of $S$ if, for every interpretation $I$ of $L$, $I$ is a model of $S$ implies that $I$ is a model for $F$.*

□

**Theorem 1** *Let $S$ be a set of closed formulas and $F$ be a closed formula of a first order language $I$. Then $F$ is a* `logical consequence` *of $S$ iff $S \cup \{\neg F\}$ is* `unsatisfiable` *or* `inconsistent`.

A set of clauses is satisfiable iff there is an interpretation that satisfies every clause which is in this set. Otherwise, it is unsatisfiable. It is generally true ([GN87]), that as one writes more clauses, the number of possible models decreases. This raises the question of whether it is possible for an individual to define his symbols so thoroughly that no interpretation is possible except the one he intended. As it turns out, there is no way in general of ensuring a unique interpretation, no matter how many clauses we write down.

**Definition 12** *Let $L$ be a first order language. The* `Herbrand universe` *$U_L$ for $L$ is the set of all ground terms, which can be formed out of the constants and function symbols appearing in $L$. The* `Herbrand base` *$B_L$ for $L$ is the set of all ground atoms which can be formed by using predicate symbols from $L$ with ground terms from the Herbrand universe as arguments.*

□

**Definition 13** *A* `substitution` *$\theta$ is any finite set of associations between variables and expressions in the form $\{v_1/t_1, \ldots, v_n/t_n\}$, where each $v_i$ is a variable, each $t_i$ is a term different from $v_i$ and the variables $v_1, \ldots, v_n$ are distinct. Each element $v_i/t_i$ is called a* `binding` *for $v_i$ and each variable is associated with at most one expression. No variable with an associated expression occurs within any*

*of the associated expressions. $\theta$ is called a* `ground substitution` *if the $t_i$ are all ground terms.*

□

**Definition 14** *Let $S$ be a finite set of expressions. A substitution $\theta$ is called a* `unifier` *for $S$ if $S\theta$ is a singleton. A unifier $\theta$ is called a* `most general unifier` *(mgu) for $S$ if, for each unifier $\sigma$ of $S$, there exists a substitution $\gamma$ such that $\sigma = \theta\gamma$*

□

## 2.2 Resolution

We only consider definite programs in this section. A resolution deduction of a clause $C$ from a program $P$ is a sequence of clauses in which (1) $C$ is an element of the sequence, and (2) each element is either a member of $P$ or the result of applying the resolution principle to clauses earlier in the sequence. The resolution principle was introduced by Robinson ([Rob65]), based on earlier work of others. It is a simple yet extremely powerful inference procedure because it uses just one rule of inference. It had been proved that resolution is sound and complete with respect to definite programs.

Typical use of resolution is in demonstrating unsatisfiability. If a set of clauses is unsatisfiable, then it is always possible by resolution to derive a contradiction from the clauses in the set. In clausal form, a contradiction takes the form of the empty clause, which is equivalent to a disjunction of no literals. Thus, to automate the determination of unsatisfiability, all we need to do is to use resolution to derive consequences from the set to be tested, terminating whenever the empty clause is generated.

Demonstrating that a set of clauses is unsatisfiable can also be used to demonstrate that a formula is logically implied by a set of formulas. Suppose we wish to show that a set of formulas $P$ logically implies a formula $F$. We can do this by finding a proof of $F$ from $P$; i.e., by establishing $P \models F$. From the refutation theorem ([Rob65]), we can establish $P \models F$ by showing that $P \cup \{\neg F\}$ is inconsistent (unsatisfiable). Thus, if we show that the set of formulas $P \cup \{\neg F\}$ is unsatisfiable, we have demonstrated that $P$ logically implies $F$.

## 2.3 Fixpoints

An elegant way of studying logic programs without negation has been proposed in [EK76]. This definition still makes sense in the presence of negation. The idea is to use a natural closure operator and equate the models of a program $P$ with the pre-fixed points of the operator, which are simpler to analyze. This operator is usually denoted by $T_P$. It maps interpretations of $P$ into interpretations of $P$ and is defined as follows:

> $A \in T_P(I)$ iff for some ground instance $A \leftarrow L_1, \ldots, L_m$ of a clause in $P$ and $\{L_1, \ldots, L_m\} \subseteq I$

Intuitively, $T_P(I)$ is the set of immediate conclusions of $I$, i.e., those which can be obtained by applying a rule from $P$ only once.

**Definition 15** *Let $S$ be a set with a partial order $\leq$. Then $a \in S$ is an* `upper bound` *of a subset $X$ of $S$ if $x \leq a$, for all $x \in X$. Similarly, $b \in S$ is a* `lower bound` *of $X$ if $b \leq x$, for all $x \in X$. $a \in S$ is the* `least upper bound` *of a subset $X$ of $S$ if $a$ is an upper bound of $X$ and, for all upper bounds $a'$ of $X$, we have $a \leq a'$. Similarly, we can define* `greatest lower bound`*. A partially ordered set $L$ is a* `complete lattice` *if the least upper bound and the greatest lower bound*

*exist for every subset $X$ of $L$.*

□

**Definition 16** *Let $L$ be a complete lattice and $T$: $L \rightarrow L$ be a mapping. We say $T$ is* `monotonic` *if $T(x) \leq T(y)$, whenever $x \leq y$. We say $a \in L$ is the* `least fixpoint` *of $T$ if $a$ is a fixpoint ($T(a) = a$) and for all fixpoints $b$ of $T$, we have $a \leq b$, Similarly, we can define* `greatest fixpoint`.

□

**Theorem 2** *Let $L$ be a complete lattice and $T$: $L \rightarrow L$ be monotonic. Then $T$ has a least fixpoint ($lfp(T)$)and a greatest fixpoint ($gfp(T)$).*

We say that $T$ is *monotonic* if $I_1 \subseteq I_2$ implies $T_P(I_1) \subseteq T_P(I_2)$. When $T_P(I) \subseteq I$ then we say that $I$ is a *pre-fixpoint* of $T$ and when $T_P(I) = I$ then we say that $I$ is a *fixpoint* of $T$. The following theorem is critical for fixpoint semantics[EK76]:

**Theorem 3** *A monotonic operator $T$ has a least fixpoint that is also the least pre-fixpoint of $T$.*

Now suppose that $T$ is one of the operators $T_P$ on Herbrand interpretations. The importance of theorem 3 stems from the fact that for a definite program $P$ the operator $T_P$ is monotonic. This is not so in the presence of negation. If $P$ is a program without negation, then $T_P$ is monotonic; the intersection of two models of $P$ is a model of $P$, and $P$ has a least model. On the other hand, if $P$ is a program with negation, then $T_P$ does not need to be monotonic; the intersection of two models of $P$ is not necessarily a model of $P$, and $P$ might have no least model.

**Theorem 4** *Let $P$ be a program. Then $I$ is a model of $P$ iff $T_P(I) \subseteq I$.*

Fixpoint models are especially interesting — particularly in the context in which negation is usually considered in database theory and logic programming. In this context, a positive ground literal is assumed false unless it can be supported in some way by the program.

## 2.4 Negation as Failure

By "negation as failure" or SLDNF-resolution, we mean the result of adding negation as failure to SLD-resolution. This is the query evaluation procedure described in [Cla78], where a negative literal is selected only if it is a ground literal $\neg q$. The next step is then to query $q$; if $q$ succeeds, then the evaluation of $\neg q$ fails; if $q$ fails on every evaluation path then, $\neg q$ succeeds. Since the evaluation tree is finite, this is called *finite failure* to distinguish it from the situation where the evaluation tree has no successful path but has infinite paths.

The different evaluation paths correspond to the different program clauses whose head matches the chosen literal. Different query evaluation procedures result from different rules for selecting the literal to resolve. These selection rules are called *computation rules.* A query *flounders* if some evaluation path ends in a goal containing only non-ground negative literals.

Negation as failure has been studied extensively as a means of extending the power of logic programming without taking on the burden of full-fledged non-Horn resolution. For a ground atom $q$, if $q$ is false in all Herbrand models of $P$ ($P$ is a normal program), then $\neg q$ is inferred and $q$ belongs to the complement (with respect to the Herbrand base $B_P$) of $gfp(T_P)$. This is a larger set than that of the atoms $l$ that are false in all models of $P$, i.e., those for which $P \models \neg l$. But it is smaller than the set of those that are false under the closed world assumption; that set is the complement of $lft(T_P)$.

SLDNF-resolution is both sound and complete with respect to the completion of a definite program (see [Llo87]).

## 2.5 Completions of Programs

Another way of adding negative information to logic programs is the completion of a program proposed by Clark ([Cla78]). His idea was to reinterpret the implications within a program as equivalences. In this way one adds to the program the "only if" part which allows one to infer negative consequences.

Formally the *completion* is defined as follows. Let $x_1, \ldots, x_k$ be some variables not appearing in a given program $P$. First, transform each clause

$$p(t_1, \ldots, t_k) \leftarrow L_1, \ldots, L_m$$

of program $P$ into

$$p(x_1, \ldots, x_k) \leftarrow \exists y_1, \ldots, y_l (x_1 = t_1), \ldots, (x_k = t_k), L_1, \ldots, L_m$$

where $y_1, \ldots, y_l$ are the variables of the original clause. Next, change each set of the transformed clauses of the form

$$\begin{aligned} p(x_1, \ldots, x_k) &\leftarrow S_1 \\ &\vdots \\ p(x_1, \ldots, x_k) &\leftarrow S_n \end{aligned}$$

where $n \geq 1$, into

$$\forall x_1, \ldots, x_k \; p(x_1, \ldots, x_k) \leftrightarrow S_1 \vee \ldots \vee S_n.$$

It is essential to include some axioms (see [Llo87] on page 79) which constrain $=$. Then the completion of $P$ is denoted by $comp(P)$.

It is often argued that in everyday language when one writes $P$ one really intends to assert $comp(P)$. The step from $P$ to $comp(P)$ is certainly a step away from the clarity in ideal logic programming, where the declarative meaning of a program should be apparent from the text of the program as written. Although in simple cases $comp(P)$ may be what most people have in mind when they write $P$, it is not easy to foresee the effect of forming $comp(P)$ when $P$ contains "recursive" clauses with the same predicate occurring on both sides of the implication sign, or clauses involving mutual recursion.

## 2.6 Stable Models

The stable model semantics is based upon the *canonical models* ([GL88]). The idea of canonical model approach is that a declarative semantics for a class of logic programs can be defined by selecting, for each program $P$ in this class, one of its models as the "canonical" model $CM(P)$. This model determines which answer to a given query is considered to be correct. For instance, a query without variables should be answered *yes* if it is true in $CM(P)$, and *no* otherwise. The canonical model is usually selected among the Herbrand models of $P$, i.e., among the models whose universe is the set of ground terms of the language of $P$, and whose object and function constants are interpreted in such a way that every ground term denotes itself.

Recall that $T_P$ is an operator defined as a mapping from one interpretation to another interpretation. All the interpretations used in this section are *2-valued.*

**Definition 17** *Let $P$ be a definite logic program and $I$ be an interpretation of $P$.*

*We define $T_P \uparrow \omega$ as follows:*

$$T_P \uparrow 0(I) = I$$
$$T_P \uparrow (n+1)(I) = T_P(T \uparrow n(I)) \bigcup T_P \uparrow n(I)$$
$$T_P \uparrow \omega(I) = \bigcup_{n=0}^{\infty} T_P(n)(I)$$

□

Suppose $P$ is a normal program, in which each rule containing variables is replaced by all its ground instances, so that all atoms in $P$ are ground, and $M \subseteq B_P$ is an interpretation. The program $P_M$ of $P$ is the logic program obtained as follows:

- Every negation-free clause in $P$ is in $P_M$.
- If $A \leftarrow B_1, \ldots, B_m, \neg D_1, \ldots, \neg D_n$ is a clause in $P$ such that for all $1 \leq j \leq n$, $D_j \notin M$, then $A \leftarrow B_1, \ldots, B_m$ is in $P_M$.
- Nothing else is in $P_M$.

M is said to be *stable* iff $M = T_{P_M} \uparrow \omega$.

**Theorem 5** *([GL88]) Any stable model of $P$ is a minimal Herbrand model of $P$.*

The intuitive meaning of stable models can be described in the same way as the intuition behind "stable expansions" in autoepistemic logic: they are "possible sets of beliefs that a rational agent might hold"[Moo85]. If $M$ is the set of ground atoms that a rational agent considers true, then any rule that has a subgoal $\neg q$ with $q \in M$ is, from agent's point of view, useless; furthermore, any subgoal $\neg q$ with $q \notin M$ is, from agent's point of view, trivial. Then the agent can simplify

the premises $P$ and replace them by $P_M$. If $M$ happens to be precisely the set of atoms that logically follow from the simplified set of premises $P_M$, then he is "rational". The stable model semantics is defined for a logic program $P$, if $P$ has exactly one stable model, and it declares that model to be the canonical model of $P$.

**Example 2.1** Consider the program $P$ consisting of the following clauses:

$$a \leftarrow \neg b$$
$$b \leftarrow \neg a.$$

Suppose $M_1 = \{a\}$, we can have $P_{M_1}$:

$$a \leftarrow$$

whose least minimal Herbrand model is $M_1$ itself. Therefore, $M_1 = T_{P_{M_1}} \uparrow \omega$. So $M_1$ is a stable model of $P$. It is easy to see that $M_2 = \{b\}$ is also a stable model of $P$.

There are two kinds of programs for which the stable model semantics is not applicable: the programs that have no stable models, and the programs that have several stable models ([GL88]). The program consisting of just one clause $a \leftarrow \neg a$ has no stable models. Given a program $P$, the completion of $P$ may have a unique Herbrand model, but $P$ may possess no stable models. The stable model semantics has a simple and elegant definition. It also extends the *perfect model semantics.* Some important drawbacks of the stable model semantics are the restrictive usage of the stable model semantics for a certain class of programs and the unintended semantics of the stable model semantics shown in ([PP93]).

## 2.7 Well-Founded Models

The key idea of the *well-founded semantics* is the concept of an *unfounded set* ([GRS91]), which is an adaptation of the "closed set" developed for disjunctive databases by [RT88]. Let us denote the complement of a set of literals $T$ by $\neg \circ T$. If $l$ is an atom, then the complement of $l$ is $\neg l$ and the complement of $\neg l$ is $l$. So $\neg \circ T = \{l \mid l$ is a complement of $l'$ and $l'$ is an arbitrary element in $T\}$. We say $A \subseteq B_P$ is an *unfounded set* (of $P$) with respect to $I$ if each atom $p \in A$ satisfies the following condition: For each instantiated clause $R$ of $P$ whose head is $p$, (at least) one of the following holds:

1. Some (positive or negative) subgoal $q$ of the body is false in $I$.
2. Some positive subgoal of the body occurs in $A$.

A literal that makes one of the above true is called a witness of unusability for clause $R$ (with respect to $I$). Intuitively, we regard $I$ as what we already know about the intended model of $P$.

**Definition 18** *([GRS91]) Let $P$ be a logic program and $I$ be a partial interpretation of $P$. The* `greatest unfounded set` *of $P$ with respect to $I$, denoted by $US_P(I)$, is the union of all sets that are unfounded with respect to $I$.*

□

Transformations $T_P, U_P$, and $W_P$ from sets of literals to sets of literals defined as follows:

1. $p \in T_P(I)$ iff there is some instantiated rule $r$ of $P$ such that $r$ has head $p$ and each literal in the body of $r$ is in $I$.

2. $U_P(I)$ is the greatest unfounded set of $P$ with respect to $I$.

3. $W_P(I) = T_P(I) \bigcup \neg \circ U_P(I)$.

Since $W_P$ is monotonic, it has a least fixpoint. We denote this least fixpoint by $W_P^\infty(I)$ and call this the *well-founded model* of $P$. Note that $W_P^\infty(I)$ is a three-valued model in the sense of [Fit85]. A ground atom may appear positively, negatively, or not all in $W_P^\infty(I)$.

The *well-founded semantics* of a program $P$ is the "meaning" represented by the least fixpoint of $W_P$, or $W_P^\infty(I)$ described above; every positive literal denotes that its atom is true, every negative literal denotes that its atom is false, and missing atoms have no truth value assigned by the semantics.

**Example 2.2** Consider the program consisting of the following clauses:

$$a \leftarrow \neg b$$
$$b \leftarrow \neg a$$
$$c \leftarrow d, \neg e$$
$$d \leftarrow c.$$

The literal set $\{c, d, e\}$ consists of an unfounded set with respect to $I = \emptyset$. Actually, $e$ is unfounded due to the condition 1: there is no clause headed with $e$ in our given program, which tells us that it is impossible to derive $e$ under any circumstances. $c$ and $d$ are unfounded due to condition 2: we have no way to prove $c$ without first prove $d$ (whether we can establish $\neg e$ to support $c$ is irrelevant for determining the unfounded set). But in order to prove $d$ we must prove $c$ first. Clearly, there is a loop among $c$ and $d$. None of them can be proved "firstly".

In contrast, the pair $a$ and $b$ does not form an unfounded set even though they depend on each other, because the only dependence is through "negation".

As soon as one of $a$ and $b$ is proved to be false, it becomes possible to prove the other is true. If both are proved to be true at once, we have inconsistency. The well-founded model of this program is $W_P^\infty(\emptyset) = \{\neg c, \neg d, \neg e\}$.

A *2-valued well-founded model* of a program $P$ refers to a well-founded model of $P$ which covers all the atoms in the Herbrand base with respect to $P$. The relationship between stable model semantics and well-founded model semantics can be established by following theorem:

**Theorem 6** *([GL88]) If $P$ has a 2-valued well-founded model, then that model is its unique stable model.*

The well-founded semantics seems to be the most adequate extension of the perfect model semantics to logic programs and avoids some drawbacks of other approaches which we have discussed.

**Example 2.3** ([GRS91]) Consider the program $P$ consisting of the following clauses:

$$a \leftarrow \neg b$$
$$b \leftarrow \neg a$$
$$c \leftarrow \neg c$$
$$c \leftarrow \neg a.$$

It has a unique stable model $M_S = \{b, c\}$. However, the unique stable model $M_S$ seems unintuitive in the view of the fact that $c$ is a consequence of the third clause (in 2-valued logic) and therefore the last clause can be considered meaningless. The first two clauses do not seem to have any reasonable 2-valued intended semantics with respect to $M_S$. Moreover, it is easy to see that it is impossible to derive $b$ from $P$ using any Horn-resolution procedure. This is because any Horn-resolution

beginning with the goal $\leftarrow b$ will reach only the first two clauses of $P$, from which $b$ cannot be derived.

In contrast, the well-founded model of $P$ is $W_P^{\infty}(\emptyset) = \emptyset$ which seems to be more intuitive than its stable model $M_S$. From the computational complexity point of view, it has been proved that the computation of well-founded model for propositional logic programs are polynomial time ([GRS91]). But Marek and Truszczynski (see page 646 in [GRS91]) have shown that for propositional logic programs to determine whether a program has a stable model is NP-complete.

# Chapter 3

# A New Proof Procedure

Horn clause logic programming can be extended to include abduction with integrity constraints. In the resulting extension of logic programs, negation as failure can be simulated by making negative conditions abducible and by imposing appropriate denials and disjunctions as integrity constraints. Eshghi and Kowalski [EK89] have given an abductive procedure as the operational semantics of abduction and have also pointed out that the stable semantics does not provide the expected semantics for abduction. Based on this operational semantics, Dung ([Dun91]) proposed another declarative semantics, called the *preferential semantics*, for logic programs containing negations.

The preferential semantics of a logic program generalizes and unifies different semantics (e.g. well-founded semantics, stable model semantics). In this chapter we introduce the preferential semantics and discuss our new proof procedure for preferential semantics in the sense of skeptical reasoning. This chapter is the main result of our thesis.

## 3.1 Preferential Semantics

The diversity of different approaches to semantics of negation suggests that there be probably no unique intended semantics for logic programs. Which semantics should be used depends on different applications. Dung [Dun91] has defined the preferential semantics for *abduction* where negation in logic programs is treated as a form of *hypotheses*. This semantics is a 3-valued semantics, in which the totality requirement characterizing the stable model semantics is replaced by a maximality condition.

A well-known and simple approach in nonmonotonic reasoning is abduction. In the simplest case, it has the form:

> From $A$ and $A \leftarrow B$
> infer $B$ as a possible "explanation" of $A$.

*Hypotheses* which are consistent with one state of a knowledge base may become inconsistent when new knowledge is added. Poole ([Poo88]) argues that abduction is preferable to nonmonotonic logics for default reasoning. In this view, defaults are hypotheses formulated within classical logic rather than conclusions derived within some form of nonmonotonic logics.

The intuitive idea is that the user supplies a set of facts known to be true, and a pool of possible hypotheses which they are prepared to accept as part of an explanation to predict the expected observations which is consistent with the facts. This *explanation* should be viewed as a "scientific theory" based on a restricted set of possible hypotheses. It is also useful to view the explanation as a scenario in which some goal is true. The user provides what is acceptable in such scenarios.

In general, the theory of abductive reasoning is based on the notion of *abduction framework* ([Poo88], [EK89], [Dun91]) defined as triples ($P$, $H$, $IC$) where $P$ is

a first order theory representing the user supplied clauses and facts, $H$ is a set of first order formulas representing the possible hypotheses, and $IC$ is a set of integrity constraints used to determine the admissible explanations.

Given an abduction framework $(P,\ H,\ IC)$, a set of hypotheses $E$ is an abductive solution for a query $Q$ iff $P \bigcup E \models Q$ and $P \bigcup E$ satisfies $IC$.

For each predicate symbol $p$ appearing in logic program $P$ we introduce a new predicate symbol $p^-$ ([Dun91]). The new predicates are called *abducible predicates.* The benefit of introducing the new predicates is that after the transforming we can treat the transformed program as a positive logic program. Atoms of the the abducible predicates are called *abducible atoms.* Ground abducible atoms are called *hypotheses.* $HY$ represents the set of all hypotheses. An *abductive program* over the language $L$ is an abduction framework $(P,\ H,\ IC)$ where

- $P$ is a definite Horn program over $L \bigcup \{p^- | p \in L\}$ with no abducible predicates appearing in the heads of its clauses,
- $IC = \{\leftarrow p(x), p^-(x) \mid p$ is a predicate symbol in $L\}$, and
- $H = HY$.

A logic program $P$ is transformed into an abductive program $P^*$ by replacing every negative literal $\neg p(t_1, \ldots, t_n)$ in each clause body by $p^-(t_1, \ldots, t_n)$. The semantics of abductive program is based on the notions of scenario and extension in [Poo88]. From now on, we suppose all the programs are given as abductive programs, i.e., we simply denote $P^*$ by $P$.

**Definition 19** *A* scenario *of an abductive program $P$ is a first order theory $P \bigcup H$ where $H$ is a subset of $HY$ such that $P \bigcup H \bigcup IC$ is consistent. An* `extension` *of an abductive program $P$ is a maximal (with respect to set inclusion) scenario of*

$P$.

□

**Definition 20** *([Dun91]) Let $P$ be an abductive program. A set of hypotheses $E$ is an* `evidence` *of an atom $q \in B_P$ ($B_P$ is the Herbrand base of $P$) with respect to $P$ if $P \cup E \models q$. A hypothesis $q^-$ is* `acceptable` *with respect to a scenario $S$ if for every evidence $E$ of $q$, $E \cup inout(S) \cup IC$ is inconsistent. Here $inout(S) = H \cup \{q \in B_P \mid S \models q\}$.*

□

We are only interested in scenarios whose hypotheses are acceptable. A scenario $S$ is *admissible* if every hypothesis accepted in $S$ is also acceptable with respect to $S$, i.e., if $q^-$ is in $S$, then $q^-$ must be acceptable with respect to $S$.

**Definition 21** *A* `preferred extension` *of an abductive program $P$ is a maximal admissible scenario of $P$. The semantics defined by the preferred extensions is called* `preferential semantics`.

□

Let $AS_P$ denote the set of all admissible scenarios of $P$. The existence of at least one preferred extension for every program $P$ is guaranteed by theorem 7.

**Theorem 7** *([Dun91]) $(AS_P, \subseteq)$ is a complete partial order, i.e. every directed subset of $AS_P$ has a least upper bound. For every admissible scenario $S$, there is at least one preferred extension $K$ such that $S \subseteq K$.*

Theorem 8 reveals the relationship between stable extensions for the stable semantics and preferential extensions for the preferential semantics.

**Theorem 8** *([Dun91]) Every stable extension is a preferred extension but not vice versa. If $P$ is a locally stratified logic program then the unique preferred extension $S$ of $P^*$ is stable and $M = \{q \mid S \models q\}$ is the unique stable model of $P$.*

Dung has proved the soundness of the Eshghi and Kowalski's abductive proof procedure with respect to the preferential semantics. Unfortunately, this proof procedure is incomplete [GMS93] because there are cases in which the procedure loops indefinitely when applied to atoms belonging to a preferred extension of the given abductive program; therefore the procedure lacks the completeness property with respect to the preferential semantics.

**Theorem 9** *(Soundness) Let us denote an abductive refutation of Eshghi and Kowalski's proof procedure for query $q$ by $([\leftarrow q], \{\}) \rightarrow ([], H)$, which means there is an abductive refutation that starts from $q$ with empty hypothesis and ends with empty goal and a hypotheses set $H$. Suppose $P$ is an abductive program, then $P \bigcup H$ is an admissible scenario and $P \bigcup H \models q$.*

## 3.2 A Proof Procedure for Skeptical Reasoning

The preferential semantics corresponds to the "maximalism" semantics where each preferred extension represents a "belief world" of an agent who tries to conclude as much as possible from an abductive program, which is considered as an incomplete knowledge base. But in many cases we want our knowledge to be "minimal" in the sense that the knowledge we believe in may be true in all the circumstances, which is captured by well-founded semantics to a certain degree. It seems that minimalism and maximalism are two main semantic intuitions for knowledge representation schemes. In the theory of nonmonotonic inheritance, these two intuitions are known as *skepticism* and *bravecism* respectively. A skeptical reasoner

refuses to draw conclusions in ambiguous situations where a brave reasoner tries to conclude as much as possible.

Since the preferential semantics has overcome some drawbacks of both the stable model semantics and the well-founded semantics ([Dun91]), in general we can expect the preferential semantics to be closer to our common sense than those two. There is little study of operational semantics for preferential semantics and only a proof procedure in the sense of brave reasoning has been given. To the best of our knowledge, no proof procedure for the preferential semantics from the skeptical reasoning point of view has been proposed. We have developed such a proof procedure which is fundamentally different from Eshghi and Kowalski's proof procedure. To simplify our technical explanation we restrict our discussion to propositional logic. It is straightforward to extend it to predicate logic for abductive programs.

**Definition 22** *Let $R$ be a computation rule. We say that $R$ is* `positivistic` *if it selects positive literals ahead of negative ones.*

□

We require that the computation rule we use is positivistic throughout this thesis.

**Definition 23** *Let $Q$ be the query $\leftarrow q$ and $R$ be a positivistic computation rule. We define the* `proof tree` *$T_Q$ for $Q$. The root of $T_Q$ is $q$. If the query $H =\!\leftarrow A'$ is any node of $T_Q$, then its children are obtained as follows:*

- *$R$ must select a positive literal if there is one in $A'$. Let $l$ be the one selected and $U_l$ be the set of clauses whose heads are $l$ in the program. The children of $H$ are obtained from the bodies of those clauses and each child corresponds*

*to one clause in the program. If $U_l = \emptyset$, then $H$ has no children and is a* `dead leaf`.

- *If a branch contains any loop, it is a dead branch.*
- *If $A'$ contains only negative subgoals, then $A'$ is a* `negative node` *(or* `negative leaf`*). Negative nodes have no children.*

□

A *branch* of $T_Q$ is an acyclic path from the root of $T_Q$. A proof tree is actually an AND-OR graph. The leaf of a proof tree is a set of literals.

**Example 3.1** Let P be the program

$$a \leftarrow b^-$$
$$b \leftarrow a^-$$
$$c \leftarrow a$$
$$c \leftarrow b$$

and $Q$ be the query $\leftarrow c$. Then the proof tree is in Figure 3.1. Both leaves are negative leaves.

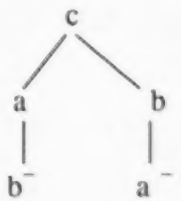


Figure 3.1: The Proof Tree of $c$

**Definition 24** *Let $P$ be a logic program and $Q$ be the query $\leftarrow q$. The* `Primary Proof Tree` *(P-tree) of $Q$ with respect to $P$ is the proof tree rooted in $q$. Let $l^-$*

*be an element of a negative leaf of a primary proof tree. The* `Assumption Proof Tree` *(A-tree) of $l$ with respect to $P$ is the proof tree rooted in $l$. While the P-tree of a query is unique, a P-tree may have many corresponding A-trees.*

□

**Example 3.2** From example 3.1 we have one primary proof tree as in Figure 3.1 and two assumption proof trees for query $\leftarrow c$ as follows:

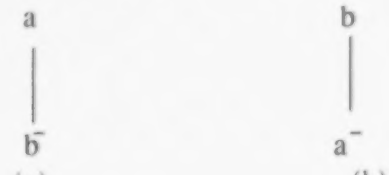


Figure 3.2: Two A-trees for $c$

**Definition 25** *A set of literals $S$ is* `consistent` *if $\forall l \in S$ we have $l^- \notin S$ and if $\forall l^- \in S$ we have $l \notin S$. Otherwise it is* `inconsistent`.

□

Recall that $W_P^\infty(I)$ is defined as a least fixpoint of $W_P$ with respect to the well-founded semantics in section 2.7.

**Definition 26** *Let $P$ be a logic program and $N$ be a set of negative literals. We say $N$ can be* `consistently assumed` *with respect to the well-founded semantics iff $W^\infty_{P\cup N}(\emptyset)$ is consistent. If $W^\infty_{P\cup N}(\emptyset)$ is inconsistent we say $N$ cannot be consistently assumed.*

□

**Definition 27** *Let $I$ be a partial interpretation of program $P$, $Q =\leftarrow q$ be a query, and $PT_Q$ be the primary proof tree of $Q$ with respect to $P$. A negative*

*leaf $N$ of $PT_Q$ is an* `active leaf` *of $PT_Q$ if the following two conditions hold simultaneously:*

- $W^{\infty}_{P \cup N}(\emptyset)$ *is consistent*
- $\forall l \in N\ W^{\infty}_{P \cup N}(\emptyset) \bigcup F$ *is inconsistent for all $F$, where $F$ is a negative leaf of the assumption proof tree $AT_{\leftarrow l}$.*

□

The first condition required for an active leaf in a P-tree guarantees that assuming all the elements in this negative leaf will not cause any inconsistency. If it results in inconsistency, we know the assumptions cannot be made for this negative leaf in any circumstances. This leaf will be useless for proving the given query. The second condition requires that any assumption we are going to make should have a *solid ground*. For example, we may have the program

$$a \leftarrow b^{-}$$
$$b \leftarrow c$$
$$c \leftarrow c^{-}$$

and let us consider query $\leftarrow a$. To establish $b$ we must first establish $c$. Before we can establish $c$ we must establish $c^{-}$. It is very clear that both $b$ and $c$ cannot be established in this program. From this we may conclude that $b^{-}$ can be established because this establishment will not bring us any trouble, i.e. $W^{\infty}_{P \cup \{b^{-}\}}(\emptyset)$ is consistent. The assumption $b^{-}$ may eligibly be made in some semantics, but both the well-founded semantics and the preferential semantics refuse to accept this assumption in their extensions. Actually, $b^{-}$ is to be considered as *undefined* (or *undetermined*) in 3-valued logics. Semantically, assuming $b^{-}$ means that $b$ must be false. Furthermore, that $b$ must be false requires that $c$ must also be false.

However, as we can see, we will get inconsistency if $c$ is false. So $b^-$ cannot be accepted in this case. With a solid ground we mean when we make an assumption (for example $b^-$) we require the existence of the complement of the evidence for the complement of this assumption ($c^-$ is the evidence of $b$ which is the complement of $b^-$, the complement of $c^-$ is $c$; therefore, we require the existence of $c$ before making assumption $b^-$).

However, an active leaf may not necessarily mean that the corresponding elements (assumptions) in this leaf can be made in the sense of skeptical reasoning. For example, consider following program:

$$a \leftarrow b^-$$
$$b \leftarrow a^-.$$

It is easy to verify that both the above two conditions are satisfied for query $\leftarrow a$, but $a$ is only in one preferential extension. There is another preferential extension $\{b, a^-\}$ which contains no $a$. So $b^-$ should not be allowed to be assumed in a skeptical reasoning context. This leads to our next definition.

**Definition 28** *Let $A$ be an active leaf of a primary proof tree. We say that $A$ is* `successful` *if either $A$ is empty (represented by $\Box$), which means all literals are resolved, or for every $l \in A$, $W^{\infty}_{P \cup F}(\emptyset)$ is inconsistent for all $F$ ($F$ is a negative leaf of the assumption proof tree $AT_{\leftarrow l}$). Otherwise, $A$ is* `failed`.

$\Box$

**Definition 29** *Let $Q$ be a query. A* `negative branch` *(or* `negative path`*) of a P-tree (or A-tree) is a branch (or path) that ends at a negative leaf. A* `dead branch` *(or* `dead path`*) of a P-tree (or A-tree) is a branch (or path) that ends at a dead leaf. An* `active branch` *(or* `active path`*) of a P-tree is a branch (or*

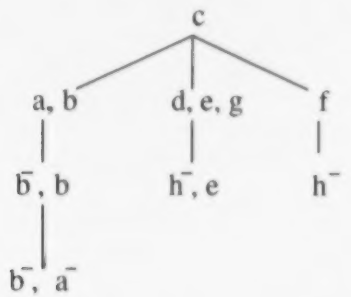


Figure 3.3: P-tree for $c$

*path) that ends at an active leaf. A* `successful branch` *(or* `successful path`*) of a P-tree is a branch (or path) that ends at a successful leaf.*

□

**Example 3.3** Let $P$ be the program

$a \leftarrow b^-$
$b \leftarrow a^-$
$c \leftarrow a, b$
$c \leftarrow d, e, g$
$c \leftarrow f$
$d \leftarrow h^-$
$f \leftarrow h^-$
$g \leftarrow$

and the query be $c$. We have the primary proof tree in Figure 3.3.

The branch $\{c\} \rightarrow \{a, b\} \rightarrow \{b^-, b\} \rightarrow \{b^-, a^-\}$ is failed because $b^-$ and $a^-$ cannot be consistently assumed. The branch $\{c\} \rightarrow \{d, e, g\} \rightarrow \{h^-, e\}$ is dead because $e$ is not negative. Only the branch $\{c\} \rightarrow \{f\} \rightarrow \{h^-\}$ is both

active ($W^{\infty}_{P\cup\{h^-\}}(\emptyset)$ is consistent and there is no clause with $h$ as head in $P$) and successful. If there is no clause with head $l$ in a given program, $l$ must be in an unfounded set according to the definition of unfounded set. This tells us that $h^-$ should be assumed in this program.

**Definition 30** *A primary proof tree P-tree is said to be* `successful` *if there is a successful branch in this P-tree. An assumption proof tree A-tree is said to be* `successful` *if there is a negative leaf L in this A-tree such that* $W^{\infty}_{P\cup L}(\emptyset)$ *is consistent.*

□

**Definition 31** *Let Q be a query for a given program P. We say the primary proof tree* $PT_Q$ *is* `inconsistent` *if* $W^{\infty}_{P\cup A}(\emptyset)$ *is inconsistent where A is the set of all active leaves of* $PT_Q$*. Otherwise,* $PT_Q$ *is consistent.*

□

**Definition 32** *Let I be a partial interpretation of program P, and* $AS_1, \ldots, AS_n$ *be negative literal sets (assumption sets) where no set is any other's subset. For any* $a_1^- \in AS_1, \ldots, a_n^- \in AS_n$*, suppose* $A_1, \ldots, A_n$ *are negative leaves of* $AT_{\leftarrow a_1}, \ldots,$ $AT_{\leftarrow a_n}$ *respectively. The* `cross consistency checks` *are to check the consistency of*

$$W^{\infty}_{P\cup A_1\cup\ldots\cup A_{i-1}\cup A_{i+1}\cup\ldots\cup A_n}(I)\cup A_i \quad (I=\emptyset)$$

*for all i* $(i = 1, \ldots, n)$ *and all the negative leaves of* $AT_{\leftarrow a_1}, \ldots, AT_{\leftarrow a_n}$

□

Let $P$ be a program and $Q = \leftarrow q$ be a query. Our proof procedure for the preferential semantics in skeptical reasoning is given below.

1. **Initial Setting**:
   Construct the P-tree $PT_Q$ rooted in query $q$. For any negative branch, let $L$ be the leaf. If $W^{\infty}_{P \cup L}(\emptyset)$ is consistent and $\forall l^- \in L$, $W^{\infty}_{P \cup L}(\emptyset) \bigcup L'$ is inconsistent for all negative leaf $L'$ of the A-tree $AT_{\leftarrow l}$, then mark this leaf *active* in $PT_Q$. Otherwise mark it *failed* in $PT_Q$. Check the consistency of the $PT_Q$ and there are two cases as in cases 2 and 3.

2. $PT_Q$ **inconsistent**:
   Suppose $PT_Q$ has $n$ ($n \geq 2$) active leaves and $L_i$ ($i = 1, \ldots, n$) is an arbitrary active leaf, make cross consistency checks. If all the checks are inconsistent, return "yes", otherwise return "no", and stop.

3. $PT_Q$ **consistent**:
   Let $L$ be an active leaf of $PT_Q$. For any $l^- \in L$, construct an A-tree $AT_{\leftarrow l}$ rooted in $l$. If there is no successful branch in $AT_{\leftarrow l}$, then mark this A-tree failed. Otherwise, mark it successful. If for all the elements in $L$, their corresponding A-trees are dead or failed, mark the branch, ended with active leaf $L$, of $PT_Q$ successful.

4. If there is any successful branch in P-tree $PT_Q$, return "yes", otherwise return "no".

We only use primary and assumption proof trees for answering a query and both P-tree and A-tree are the same syntactically, although their semantics are different. This two level of proof trees strategy will bring many benefits to the implementation aspects. Our initial setting will guarantee that we only explore those leaves which are active, because only an active leaf can contribute to a successful proof. Under those two conditions in the initial setting, there is a possibility that $q$ might be in all extensions. The inconsistency of $W^{\infty}_{P \cup L}(\emptyset) \bigcup L'$ allows us to assume $L$ with explicit evidence from $L'$. This prevents us from

making assumptions without a solid ground. For example, we may assume $l^-$ in an extension $E$, if $l \notin E$. Obviously this violates the definitions of both the well-founded semantics and the preferential semantics because $l$ may be undefined in $E$. $l^-$ can be assumed only if we have *explicit evidence* blocking $l$ to be true.

The inconsistency of a P-tree means that the hypotheses are conflicting, so there could exist more than one extension. In this situation we want to know whether there is an extension not containing $q$. Obviously, the fact that $q$ is not included in an extension tells us that no active leaf of $q$ is admissible to this extension with respect to a given logic program. If such an extension exists, it must contain a set of literals such that this set blocks all the active leaves in the P-tree of $q$. In case this kind of set is found, we know, for sure, the query cannot be in every preferential extension. Otherwise we can answer the query with "yes". This is because it is guaranteed that there is at least one successful branch in P-tree for any extension. A typical example is Example 3.1. For query $\leftarrow c$, the P-tree is inconsistent. We need to do cross consistency checks. If there is an extension that contains $a$ and $b$, we know $c$ will never succeed in this extension. In order to establish $a$ (or $b$ respectively) we must establish $b^-$ (or $a^-$ respectively) first. Therefore, the extension containing both $a$ and $b$ must also include both $b^-$ and $a^-$. Our cross consistency checks will eliminate this case by recognizing that accepting $b^-$ and $a^-$ will result in inconsistency.

When a P-tree is consistent, we should try every possibly successful branch. For each active branch we check every element of its leaf node. If there is no evidence against this element (i.e. the corresponding A-tree fails), it succeeds (i.e. it can be consistently assumed). The failure of an A-tree $AT_{\leftarrow l}$ guarantees that no extension contains any literal $l$. Therefore, some active leaf must be admissible to any preferential extensions by the definition of the preferential extension. This is exactly the same idea behind the preferential semantics. The admissibility of an active leaf gives us that the query should be included.

## 3.3 Examples

In this section we will give some examples to see how our proof procedure works and to provide a better understanding of the definitions given in last section.

**Example 3.4** Let P be the program

$$a \leftarrow b^-$$
$$b \leftarrow a^-$$
$$c \leftarrow a$$
$$c \leftarrow b.$$

Then $P$ has two preferential extensions $E1 = \{a, c, b^-\}$ and $E1 = \{b, c, a^-\}$. The well-founded semantics says the extension (model) of $P$ is empty. We have seen that $c$ is contained by both $E1$ and $E2$. For query $\leftarrow c$, we have the P-tree in Figure 3.1. It is easy to see that $W^{\infty}_{P\cup\{a^-\}}(\emptyset)$ and $W^{\infty}_{P\cup\{b^-\}}(\emptyset)$ are consistent respectively. But $W^{\infty}_{P\cup\{a^-\}}(\emptyset)\bigcup\{b^-\}$ and $W^{\infty}_{P\cup\{b^-\}}(\emptyset)\bigcup\{a^-\}$ are inconsistent. So both negative leaves of $PT_{\leftarrow c}$ are active leaves. Since $W^{\infty}_{P\cup\{a^-,b^-\}}(\emptyset)$ is inconsistent (from $b^-$ we can derive $a$, so both $a$ and $a^-$ are in $W^{\infty}_{P\cup\{a^-,b^-\}}(\emptyset)$), the P-tree $PT_{\leftarrow c}$ is inconsistent. According to the procedure, the cross consistency check should be performed next. Only two cross consistency checks are needed in this case because there is only one element in every active leaf of $PT_{\leftarrow c}$. Since the active leaf for $a$ is $b^-$ in $AT_{\leftarrow a}$ and the active leaf for $b$ is $a^-$ in $AT_{\leftarrow b}$, we have $W^{\infty}_{P\cup\{a^-\}}(\emptyset)\bigcup\{b^-\}$ and $W^{\infty}_{P\cup\{b^-\}}(\emptyset)\bigcup\{a^-\}$ are inconsistent. Therefore, the answer for query $c$ is "yes".

As for query $\leftarrow a$, part (a) of Figure 3.2 is the P-tree $PT_{\leftarrow a}$ and part (b) of Figure 3.2 is the A-tree $AT_{\leftarrow b}$. Since $W^{\infty}_{P\cup\{b^-\}}(\emptyset)$ is consistent and there is only one branch in the $PT_{\leftarrow a}$, the $PT_{\leftarrow a}$ is consistent. We need to check the consistency of the A-tree. From the fact that $W^{\infty}_{P\cup\{a^-\}}(\emptyset)$ is consistent, we conclude that $b^-$ cannot be assumed in all circumstances. Therefore, the answer for query $a$ is "no".

Similarly we can get the answer "no" for query $b$.

**Example 3.5** Let P be the program

$$a \leftarrow b^-$$
$$b \leftarrow c^-$$
$$c \leftarrow a^-$$

and $Q$ be the query $\leftarrow a$. We have the P-tree and A-tree as follows:

Figure 3.4: (a) P-tree of $a$ (b) A-tree of $b$

Since $W^{\infty}_{P\cup\{b^-\}}(\emptyset) = \{a, b^-\}$ and $W^{\infty}_{P\cup\{b^-\}}(\emptyset)\cup\{c^-\} = \{a, b^-, c^-\}$ are both consistent, the only negative leaf of $PT_{\leftarrow a}$ is not an active leaf. So we conclude that the answer for the query $a$ is "no". Similarly we can get the same answers for $b$ and $c$. As a matter of fact, there is only one preferential extension which is empty. This coincides with the well-founded semantics.

**Example 3.6** Let P be the program

$$a \leftarrow b^-$$
$$b \leftarrow c$$
$$c \leftarrow c^-$$

The only preferential extension for $P$ is empty. Let us look at query $\leftarrow a$. Considering that both $W^{\infty}_{P\cup\{b^-\}}(\emptyset) = \{a, b^-\}$ and $W^{\infty}_{P\cup\{b^-\}}(\emptyset)\cup\{c^-\} = \{a, b^-, c^-\}$

are consistent, the only negative leaf of $PT_{\leftarrow a}$ is not an active leaf. So we conclude that the answer for the query $a$ is "no". Here $b^-$ cannot be assumed because the complement of $b^-$ is the $b$ and $b$'s evidence is $c^-$, and we do not have the complement of $c^-$ in our extension. In this case we say that there is no *explicit evidence* blocking $b$ from being true.

As for the answers of $b$ and $c$, both negatives in $PT_{\leftarrow b}$ and $PT_{\leftarrow c}$ are failed respectively because they are inconsistent. Consequently, the answers are "no" for both $b$ and $c$. Both the preferential extension and the well-founded model are empty for $P$.

**Example 3.7** Let P be the program

$$a \leftarrow b^-$$
$$b \leftarrow a,\ a^-$$

and $Q$ be the query $\leftarrow a$. We have the P-tree and A-tree as in Figure 3.5:

Figure 3.5: (a) P-tree $PT_{\leftarrow a}$ (b) A-tree $PT_{\leftarrow b}$

Since $W^{\infty}_{P\cup\{b^-\}}(\emptyset)$ is consistent and $W^{\infty}_{P\cup\{b^-\}}(\emptyset)\cup\{a^-, b^-\} = \{a, a^-, b^-\}$ is inconsistent, the only branch in $PT_{\leftarrow a}$ is an active branch. Because of the inconsistency of $W^{\infty}_{P\cup\{a^-,b^-\}}(\emptyset)$ the answer for the query $a$ is "yes". We can also conclude that the answer for the query $b$ is "no" because there is no active leaf in P-tree $PT_{\leftarrow b}$. Actually there is only one preferential extension which is $\{a, b^-\}$. Although its well-founded model is empty.

**Example 3.8** Let P be the program

$$a \leftarrow b^-$$
$$b \leftarrow a,\ c^-$$
$$d \leftarrow c^-$$

and $Q$ be the query $\leftarrow a$. We have the P-tree and A-tree in Figure 3.6.

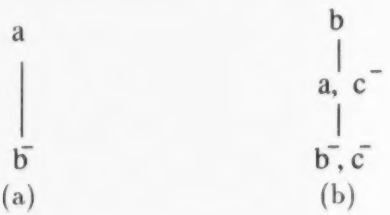


Figure 3.6: (a) P-tree $PT_{\leftarrow a}$ (b) A-tree $AT_{\leftarrow b}$

On account of the inconsistency of $W^{\infty}_{P\cup\{b^-\}}(\emptyset) = \{a, b, d, b^-, c^-\}$, there is no active leaf in $PT_{\leftarrow a}$. So the answer for the query $a$ is "no". The only preferential extension is $\{d, c^-\}$. This is simply because $c^-$ must be acceptable to any scenario due to the fact that there is no clause headed with $c$ in $P$. From $c^-$ we have $d$.

**Example 3.9** Let $P$ be the program

$$a \leftarrow b^-$$
$$a \leftarrow c^-$$
$$b \leftarrow a^-$$
$$c \leftarrow d^-$$
$$d \leftarrow a^-.$$

There is only one preferential extension $E = \{a, c, b^-, d^-\}$ for $P$. The P-tree for $\leftarrow a$ is shown in Figure 3.7.

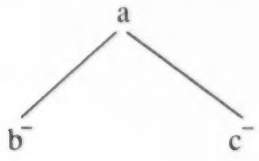


Figure 3.7: P-tree for $a$

Since $W^{\infty}_{P\cup\{b^-\}}(\emptyset) = \{a, c, b^-, d^-\}$ is consistent and $W^{\infty}_{P\cup\{b^-\}}(\emptyset)\cup\{a^-\}$ $= \{a, c, a^-, b^-, d^-\}$ is inconsistent, the branch $\{a \rightarrow b^-\}$ is an active branch. But $W^{\infty}_{P\cup\{c^-\}}(\emptyset) = \{a, c, b^-, c^-, d^-\}$ is inconsistent, so the branch $\{a \rightarrow c^-\}$ is a failed branch. Hence, the P-tree $PT_{\leftarrow a}$ is consistent. Our next step is to check the consistency of A-tree $AT_{\leftarrow b}$. Due to the inconsistency of $W^{\infty}_{P\cup\{a^-\}}(\emptyset) = \{a, b, d, a^-, c^-\}$, $AT_{\leftarrow b}$ is failed. $b^-$ can be consistently assumed. Consequently, the answer for query $\leftarrow a$ is "yes".

For query $\leftarrow c$, $W^{\infty}_{P\cup\{d^-\}}(\emptyset) = \{a, c, b^-, d^-\}$ is consistent and $W^{\infty}_{P\cup\{d^-\}}(\emptyset)\cup\{a^-\}$ $= \{a, b, d, a^-, c^-\}$ is inconsistent. For the reason that $PT_{\leftarrow c}$ is consistent we need to check the consistency of A-tree $AT_{\leftarrow a}$. From the inconsistency of $W^{\infty}_{P\cup\{a^-\}}(\emptyset) =$ $\{a, b, d, a^-, c^-\}$ we have the answer for $c$ is "yes" too. As for the query $\leftarrow b$, there is only one negative leaf in $PT_{\leftarrow b}$. This negative leaf is not an active leaf by the fact that $W^{\infty}_{P\cup\{b^-\}}(\emptyset) = \{a, b, d, a^-, c^-\}$ is inconsistent. Accordingly, the answer for query $\leftarrow b$ is "no".

**Example 3.10** Let $P$ be the program

$a \leftarrow b^-, d^-$
$b \leftarrow a^-, d^-$
$c \leftarrow a$
$c \leftarrow b$
$d \leftarrow e^-$
$e \leftarrow d^-$.

There are three preferential extensions: $E1 = \{a, b, e, d^-\}$, $E2 = \{a, c, e, d^-\}$, $E3 = \{d, a^-, b^-, e^-\}$. The well-founded model is empty. For query $\leftarrow c$ the P-tree $PT_{\leftarrow c}$ is shown in Figure 3.8.

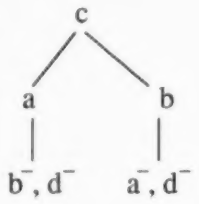


Figure 3.8: P-tree $PT_{\leftarrow c}$ for query $\leftarrow c$

By the consistency of $W^{\infty}_{P\cup\{b^-,d^-\}}(\emptyset) = \{a, c, e, d^-\}$, the inconsistency of $W^{\infty}_{P\cup\{b^-,d^-\}}(\emptyset)\cup\{a^-, d^-\} = \{a, c, e, a^-, d^-\}$, as well as the inconsistency of $W^{\infty}_{P\cup\{b^-,d^-\}}(\emptyset)\cup\{e^-\} = \{a, c, e, d^-, e^-\}$, we know that the negative leaf $\{b^-, d^-\}$ of $PT_{\leftarrow c}$ is an active leaf. Also by the consistency of $W^{\infty}_{P\cup\{a^-,d^-\}}(\emptyset) = \{b, c, e, a^-, d^-\}$, the inconsistency of $W^{\infty}_{P\cup\{a^-,d^-\}}(\emptyset)\cup\{b^-, d^-\} = \{b, c, e, a^-, b^-, d^-\}$, and the inconsistency of $W^{\infty}_{P\cup\{a^-,d^-\}}(\emptyset)\cup\{e^-\} = \{b, c, e, a^-, d^-, e^-\}$, we know that the negative leaf $\{b^-, d^-\}$ of $PT_{\leftarrow c}$ is an active leaf. $PT_{\leftarrow c}$ is inconsistent for the sake of inconsistency of $W^{\infty}_{P\cup\{a^-,b^-,d^-\}}(\emptyset) = \{a, b, c, a^-, b^-, d^-\}$. Hence, we need to do the cross consistency check. The cross consistency checks of $\{b^-, a^-\}$ are as follows:

$$W^{\infty}_{P\cup\{a^-,d^-\}}(\emptyset)\cup\{b^-, d^-\} = \{b, c, e, a^-, b^-, d^-\} \text{ is inconsistent}$$
$$W^{\infty}_{P\cup\{b^-,d^-\}}(\emptyset)\cup\{a^-, d^-\} = \{a, c, e, a^-, b^-, d^-\} \text{ is inconsistent}$$

Furthermore, the cross consistency checks of $\{b^-, d^-\}$ are as follows:

$$W^{\infty}_{P\cup\{a^-,d^-\}}(\emptyset)\cup\{e^-\} = \{b, c, e, a^-, d^-, e^-\} \text{ is inconsistent}$$
$$W^{\infty}_{P\cup\{e^-\}}(\emptyset)\cup\{a^-, d^-\} = \{d, a^-, d^-, e^-\} \text{ is inconsistent}$$

Similarly, we can have the same result for the cross consistency checks of $\{d^-, a^-\}$. However, the cross consistency check of $\{d^-, d^-\}$:

$$W^{\infty}_{P\cup\{e^-\}}(\emptyset)\bigcup\{e^-\} = \{d, a^-, b^-, e^-\}$$

is consistent. In consequence, the answer for the query $\leftarrow c$ is "no".

**Example 3.11** Let P be the program

$$\begin{aligned} c &\leftarrow a^- \\ c &\leftarrow b^- \\ a &\leftarrow b^- \\ b &\leftarrow a^- \\ c &\leftarrow d^- \\ d &\leftarrow c^- \end{aligned}$$

There are two preferential extensions: $E1 = \{a, c, b^-, d^-\}$ and $E2 = \{b, c, a^-, d^-\}$. However, the well-founded model is empty. First let us see the answer for the query $\leftarrow c$. Figure 3.9 is the P-tree for $c$.

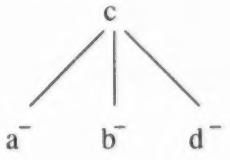


Figure 3.9: P-tree $PT_{\leftarrow c}$ for query $\leftarrow c$

With regard to the negative leaf $a^-$: $W^{\infty}_{P\cup\{a^-\}}(\emptyset) = \{b, c, a^-, d^-\}$ is consistent, whereas $W^{\infty}_{P\cup\{a^-\}}(\emptyset)\cup\{b^-\} = \{b, c, a^-, b^-, d^-\}$ is inconsistent. Therefore, the leaf $a^-$ is an active leaf. As for the negative leaf $b^-$: $W^{\infty}_{P\cup\{b^-\}}(\emptyset) = \{a, c, b^-, d^-\}$ is consistent, whereas $W^{\infty}_{P\cup\{b^-\}}(\emptyset)\cup\{a^-\} = \{a, c, a^-, b^-, d^-\}$ is inconsistent. We

know leaf $b^-$ is an active leaf. Finally, for the negative leaf $d^-$: $W^{\infty}_{P\cup\{d^-\}}(\emptyset) = \{c, d^-\}$ is consistent and $W^{\infty}_{P\cup\{d^-\}}(\emptyset)\cup\{c^-\} = \{c, c^-, d^-\}$ is inconsistent. So leaf $d^-$ is also an active leaf.

Our next step is to do the cross consistency checks:

$$W^{\infty}_{P\cup\{a^-,b^-\}}(\emptyset)\cup\{c^-\} = \{a, b, c, a^-, b^-, c^-, d^-\} \text{ inconsistent}$$
$$W^{\infty}_{P\cup\{a^-,c^-\}}(\emptyset)\cup\{b^-\} = \{b, c, d, a^-, b^-, c^-\} \text{ inconsistent}$$
$$W^{\infty}_{P\cup\{b^-,c^-\}}(\emptyset)\cup\{a^-\} = \{a, c, d, a^-, b^-, c^-\} \text{ inconsistent}$$

So the answer for query $\leftarrow c$ is "yes".

With regard to the query $\leftarrow a$, the only leaf of P-tree $PT_{\leftarrow a}$ is an active leaf for the reason that $W^{\infty}_{P\cup\{b^-\}}(\emptyset) = \{a, c, b^-, d^-\}$ is consistent and $W^{\infty}_{P\cup\{b^-\}}(\emptyset)\cup\{a^-\}$ is inconsistent. Therefore, $PT_{\leftarrow a}$ is consistent. But $W^{\infty}_{P\cup\{a^-\}}(\emptyset) = \{b, c, a^-, d^-\}$ is consistent, The answer for query $\leftarrow a$ is "no". Similarly we have the same result for query $\leftarrow b$ and $\leftarrow d$.

## 3.4 Implementations

Now we give more details about our proof procedure.

**Skeptical Reasoning Algorithm for Preferential Semantics**

Let $I$ be a partial interpretation of a given logic program $P$ and $Q = \leftarrow q$ be a query. If this algorithm returns "yes", $Q$ can be proved ($Q$ is in all the preferential extensions).

**Begin**

**Find** all the clauses unifiable with $q$;
**If** no such clause available **Then**
    **Return** "no";
**Build** a P-tree rooted in $q$ $PT_Q$**Until**:
    **Either** we encounter negative literal,
    **Or** positive literal without unifiable clause;
**If** there is any successful branch **Then**
    **Return** "yes";
**For** each negative leaf $L$ in $PT_Q$ **Do**
  **If** $W^{\infty}_{P \cup L}(\emptyset)$ inconsistent **Then**
    **Mark** $L$ failed in $PT_Q$;
    **For** each element $l^- \in L$ **Do**
      **Build** an A-tree $AT_{\leftarrow l}$ for $l$;
      **For** each negative leaf $F$ in $AT_{\leftarrow l}$ **Do**
        **If** $W^{\infty}_{P \cup L}(\emptyset) \bigcup F$ consistent **Then**
          **Mark** the leaf $L$ failed in $PT_Q$;
      **End-For**
    **End-For**
**End-For**
**If** no active leaves **Then**
  **Return** "no";
**If** $PT_Q$ inconsistent **Then**
  **For** each active leaf $L$ in $PT_Q$ **Do**
    **For** each element $l^-$ in $L$ **Do**
      **Build** an A-tree rooted in $l$;
    **End-For**;
  **End-For**;
  **If** all cross consistency checks are inconsistent **Then**

```
            Return "yes";
        Else
            Return "no";
    Else
        For each active leaf L in PT_Q Do
            For each element l⁻ in L Do
                Build an A-tree AT_←l;
                For each negative leaf F of AT_←l Do
                    If W^∞_{P∪F}(∅) inconsistent Then
                        Mark F failed in AT_←l;
                End-For;
                If all the leaves in AT_←l are failed Then
                    Mark AT_←l failed;
            End-For;
            If all the AT_←l are failed Then
            Mark L successful;
            Return "yes";
        End-For;
    End-If;
    Return "no";
End
```

We have an unfinished implementation of our proof procedure in Quintus-Prolog. Our implementation is originally for preferential answer set semantics ([YLY93]) and it is not difficult to change this implementation into an implementation for preferential semantics. We expect this could be done at the end of this July. Some ideas are borrowed from XTheorist ([Goe92]) in our implementation.

# Chapter 4

# Some Theoretical Results and Analyses

We have described our proof procedure for the preferential semantics in skeptical reasoning and from the examples in the last chapter we have also seen that our algorithm appears to coincide with the preferential semantics. We will present some theoretical results about our proof procedure and then compare it with some other approaches. The most important result is the soundness (correctness) of our algorithm with respect to the preferential semantics.

## 4.1 Soundness

**Lemma 1** *Let $P$ be a given logic program, $I$ be a partial interpretation of $P$, $Q =\leftarrow q$ be a query, and $L_1, \ldots, L_N$ $(N \geq 2)$ be active leaves of the primary proof tree $PT_Q$. If $PT_Q$ is inconsistent and the cross consistency checks are inconsistent, then $q$ must be in every preferential extension of $P$.*

**Proof**: We prove it by using induction on the number of active leaves of $PT_Q$. Suppose there is a preferential extension $E$, such that $q \notin E$.

*Base Step*: $N = 2$. The primary proof tree $PT_Q$ is shown in Figure 4.1:

Figure 4.1: P-tree for $\leftarrow q$ when $N = 2$

In Figure 4.1, $L_1 = \{l_{11}^-, l_{12}^-, \ldots, l_{1m}^-\}$ and $L_2 = \{l_{21}^-, l_{22}^-, \ldots, l_{2n}^-\}$ are two active leaves. Let us consider literal $q^-$. For $q^-$ there is only two possibilities: either $q^- \in E$ or $q^- \notin E$. We will prove that in both cases we will get a contradiction. Consequently, our assumption $q \notin E$ is false.

Suppose $q^- \in E$, from this we know that $q^-$ is acceptable with respect to $E$. However, $L_1$ and $L_2$ are the two evidence of $q$ and by the definition of preferential extensions we have both $E \bigcup L_1$ and $E \bigcup L_2$ are inconsistent. This means that $\exists i$ $(i = 1, \ldots, m)$ and $\exists j$ $(j = 1, \ldots, n)$ such that $l_{1i} \in E$ and $l_{2j} \in E$. Let $l_{1i}$ and $l_{2j}$ have the following assumption trees:

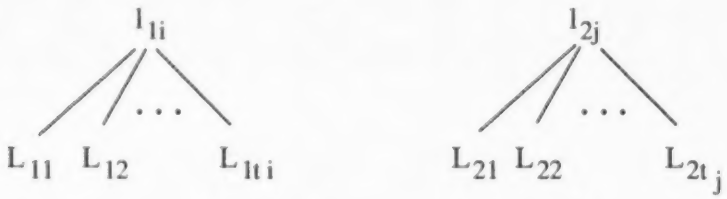


Figure 4.2: Two A-trees for $\leftarrow l_{1i}$ and $\leftarrow l_{2j}$ respectively

Since $l_{1i} \in E$, there exists $i'$ $(i' = 1, \ldots, t_i)$ such that $L_{1i'} \subseteq E$. So does $\exists j'$ and $L_{2j'} \subseteq E$. From $E = W_{P \cup E}^{\infty}(\emptyset)$, we have $E = W_{P \cup E \cup L_{1i'} \cup L_{2j'}}^{\infty}(\emptyset)$. Our cross consistency checks say that $W_{P \cup L_{1i'}}^{\infty}(\emptyset) \bigcup L_{2j'}$ is inconsistent and from this

inconsistency we can easily derive that $W^{\infty}_{P \cup L_{1i'} \cup L_{2j'}}(\emptyset)$ is also inconsistent. Hence, we have $W^{\infty}_{P \cup E}(\emptyset)$ is inconsistent and $E$ is inconsistent. Therefore, $E$ is not a preferential extension of $P$, which is contradictory to our assumption.

Suppose $q^- \notin E$. There are three cases we need to deal with:

*Case 1*: $E \bigcup L_1$ is consistent and $E \bigcup L_2$ is inconsistent. From the inconsistency of $E \bigcup L_2$, we have $\exists j$ $(j = 1, \ldots, n)$ such that $l_{2j} \in E$. Therefore, we still can find a $j'$ $(j' = 1, \ldots, t_j)$ such that $L_{2j'} \subseteq E$. But the cross consistency checks tell us that $W^{\infty}_{P \cup L_{2j'}}(\emptyset) \bigcup L_{1i'}$ for all $i'$ $(i' = 1, \ldots, t_i)$ is inconsistent. So $E \bigcup L_{1i'}$ should be also inconsistent for all $i'$. $l^-_{1i}$ should be acceptable with respect to $E$. Since our $i$ is arbitrary, we can extend this result to all the element in $L_1$. That is $L_1$ should be admissible to $E$. Then $q \in E$.

*Case 2*: $E \bigcup L_1$ is inconsistent and $E \bigcup L_2$ is consistent. This is a symmetric situation as in case 1.

*Case 3*: both $E \bigcup L_1$ and $E \bigcup L_2$ are consistent. Let $F_1 = W^{\infty}_{P \cup E \cup L_1}(\emptyset)$ and $F_2 = W^{\infty}_{P \cup E \cup L_2}(\emptyset)$. Then $F_1$ cannot be consistency. Suppose $F_1$ is consistent. As a result of $L$ being an active leaf we have $W^{\infty}_{P \cup L_1}(\emptyset) \bigcup \{$ any negative leaf of A-tree $AT_{\leftarrow l_{1i}}\}$ is inconsistent, for all $i = 1, \ldots, m$. Of course, $W^{\infty}_{P \cup E \cup L_1}(\emptyset) \bigcup \{$ any negative leaf of A-tree $AT_{\leftarrow l_{1i}}\}$ is also inconsistent for all $i = 1, \ldots, m$. $L_1$ must be admissible to $E$. Accordingly, we have $q \in E$, which is contradictory to $q \notin E$. So $F_1$ cannot be consistent. Similarly, $F_2$ is inconsistent too. Since $F_1$ (also $F_2$) is inconsistent, there is an element $l$ in Herbrand base of program $P$ such that both $l \in F_1$ and $l^- \in F_1$. We discuss $l$ for three different situations:

1. $l \in E$: The only chance for $l^- \in F_1$ is $l^- \in L_1$ because $E$ is consistent. But this is contradictory to our assumption that $E \bigcup L_1$ is consistent. Obviously, the same result will be applied to $F_2$.

2. $l^- \in E$: For every negative leaf $L$ of the A-tree $AT_{\leftarrow l}$, we have $E \bigcup L$ is inconsistent. This guarantees that every active branch of A-tree $AT_{\leftarrow l}$ cannot succeed with respect to $E$. So no matter what assumptions you add into $P$ (i.e. no matter what $L_1$ could be), $W^{\infty}_{P \cup E \cup L_1}(\emptyset)$ cannot conclude $l$ which is contradictory to the assumption $l \in F_1$. It is trivial to get the same result for $F_2$.

3. $l \notin E$ and $l^- \notin E$: By the fact that we conclude $l$ from $F_1$, for sure $l^- \in L_1$. There must exist a negative leaf in A-tree $AT_{\leftarrow l}$, as in Figure 4.3,

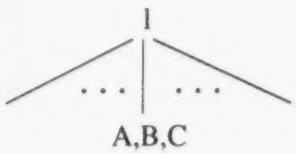


Figure 4.3: A-tree for $AT_{\leftarrow l}$

with the conditions: $A \subseteq E$, $B \subseteq L_1$, $C$ is admissible to $E \bigcup L_1$ ($C$ may be empty).

Since $L_1$ is an active leaf, we have that $W^{\infty}_{P \cup L_1}(\emptyset) \bigcup A \bigcup B \bigcup C$ is inconsistent and both $W^{\infty}_{P \cup L_1}(\emptyset)$ and $W^{\infty}_{P \cup L_1 \cup C}(\emptyset)$ are consistent. Therefore, the inconsistency can only be caused by some $p \in W^{\infty}_{P \cup L_1}(\emptyset)$ with $p^- \in A \subseteq E$. So we have $W^{\infty}_{P \cup L_1}(\emptyset) \bigcup E$ is inconsistent. Similarly we have $W^{\infty}_{P \cup L_2}(\emptyset) \bigcup E$ is inconsistent. Therefore $q^- \in E$ which is contradictory to our assumption that $q^- \notin E$.

Induction step: Suppose $N = k$, the lemma holds. We prove our result when $N = k + 1$. Let $L_i = \{l^-_{i1}, l^-_{i2}, \ldots, l^-_{it_i}\}$ $(i = 1, 2, \ldots, k+1)$. If there exists any $l_{ij} \in E$ $(i = 1, \ldots, k+1;\ j = 1, \ldots, t_i)$, then we know that the active leaf $L_i$ is a failed leaf. So the P-tree $PT_{\leftarrow q}$ has at most $k$ active leaves left. In this case, our lemma holds according to the induction hypothesis.

If for all $i$ and $j$, $l_{it} \notin E$, then all $E \bigcup L_1$, $E \bigcup L_2$, ..., $E \bigcup L_{k+1}$ are consistent. So $q^- \notin E$. A very similar proof, as in case 3 of our base step, can be constructed in a straightforward manner.

□

**Theorem 10** *(Soundness) Let $P$ be a given logic program, and $I$ be a partial interpretation of $P$. Suppose $Q = \leftarrow q$ is a query and $PT_Q$ is the primary proof tree of $\leftarrow q$. For our proof procedure:*

*If the answer for $q$, given by our proof procedure, is "yes", then $q$ is in every preferential extension of $P$.*

**Proof**: There are three cases in our proof procedure, so we discuss them separately.

- Case 1: The P-tree $PT_Q$ has a successful branch without further exploring any A-tree. In this case we have

$$q \in \bigcup_{k=1}^{\infty} T_P \uparrow k(P) \subseteq E,$$

  where $E$ is an arbitrary preferential extension of program $P$.

- Case 2: The P-tree $PT_Q$ is consistent. In this case we know that $PT_Q$ has a successful branch. Let $L$ be the leaf of this successful branch. It follows that $W^{\infty}_{P \cup L}(\emptyset)$ is consistent which satisfies the integrity constraints (see [Dun91]). For any $l^- \in L$, the A-tree $AT_{\leftarrow l}$ is failed (i.e. no leaf in $AT_{\leftarrow l}$ can be consistently assumed). This guarantees that no extension will contain $l$ which could block $L$ to be consistently assumed. That $\forall l \, \forall L' \, W^{\infty}_{P \cup L}(\emptyset) \bigcup L'$ must be inconsistent can eliminate those assumptions made simply by without the presence of positive counterpart, where $L'$ is a negative leaf of assumption tree $AT_{\leftarrow l}$. The complement of these assumptions is exactly a set of those

elements which are undefined in the well-founded semantics. From the definition of admissibility (see the section 3.1), $l^-$ must be admissible to any preferential extension. Further more, $L$ is an admissible set to any preferential extension. Therefore, $q$ must be true in every preferential extension of $P$.

- Case 3: The P-tree $PT_Q$ is inconsistent. The key idea of our algorithm is trying to find an extra extension which contains a literal set such that this set will block all the active leaves of $PT_Q$, i.e., no active leaf could be succeeded. The lemma we just proved is exactly the case as here.

□

There is little study of operational semantics for the preferential semantics and only a very efficient proof procedure, in the sense of brave reasoning, is known (see [EK89]). Unfortunately, this proof procedure cannot be used for computing extensions under the skeptical reasoning. The reason is that for a given program and a query, once there exists an extension containing this query, the proof procedure always can find it. But it is impossible for it to tell whether other extensions could contain this query or not. During the proof, the procedure may go to other extensions to try to find "evidence" without noticing that it is been in a different extension. As to our knowledge no proof procedure for the preferential semantics in the view of skeptical reasoning has known yet. So our method is a first attempt.

The closest method we have found to our algorithm is Ross's proof procedure for the well-founded semantics in [Ros92], which uses the so-called *global SLS-resolution*. SLS-resolution ([Prz89b]) is a top-down proof procedure that uses an extension of SLD-resolution to answer queries. Global SLS-resolution is an extended version of SLS-resolution. But our method is fundamentally different from Ross's. Since Dung ([Dun91]) has proved that the preferential semantics can capture the well-founded semantics, our proof procedure can be considered more

powerful in this sense.

## 4.2 Completeness and Computational Complexity

As of now we do not know whether our proof procedure is complete or incomplete. Further research is needed to investigate the completeness of this proof procedure. Our conjecture is that this procedure is complete based on its principle. The main difficulty in proving the completeness of our proof procedure comes from the lack of a constructive definition for the preferential extensions. For this reason we should reformalize the definition of an extension in the preferential semantics by finding a new constructive definition. Ross has designed a sound and complete proof procedure for the well-founded semantics ([Ros92]). The method, which he used for proving the completeness, has provided us a very good guidance for our further research.

A natural question concerning our proof procedure is, of course, the computational complexity. Contrary to our intuition, finding an extension is easier than determining whether a predicate is a member of some extension (membership) in nonmonotonic reasoning area generally (see [KS91]). The difficulty in devising sound, complete and tractable algorithms for skeptical reasoning has led many researchers to suppose that any formulation of reasoning based on an intersection of extensions is intractable [KS91]. Although this is not proved, it is a reasonable conjecture. Unfortunately, our proof procedure is intractable, i.e. the computational complexity of our algorithm is not polynomial. The number of cross consistency checks could lead to exponential times theoretically.

Consistency checks are usually considered to be expensive. Fortunately, our special consistency check is tractable for propositional logic. Since the compu-

tation of an unfounded set is polynomial time ([GRS91]) for propositional logic program and the cost for determining whether $W^{\infty}_{P \cup N}(\emptyset)$ contains a literal is linear ($N$ is a set of hypotheses) in the length of the logic program ( see [DG84]). The length of a logic program is defined as the total number of occurrences of literals in the program. So our consistency check costs polynomial time for propositional logic program.

# Chapter 5

# Conclusions

How to treat negation in a logic program is a difficult topic which has received a lot of attention in recent years in the logic programming area. Many different semantics have been proposed for a normal logic program. Dung's preferential semantics has shown some advantages over others and this is why we choose this semantics to start our research.

We have presented a new, possibly the first, proof procedure for the preferential semantics from the skeptical reasoning point of view. This proof procedure uses a different method from all known ones. The key idea of our method is to define two different proof trees, called *Primary Proof Tree* and *Assumption Proof Tree* respectively. By checking the consistency of those two proof trees' leaves with respect to the well-founded semantics, we can answer a query correctly for a given program. The strategies we have used in our proof procedure are both bottom-up and top-down. We have also shown the soundness of our method with respect to the preferential semantics.

Further research could be to investigate the completeness of this proof procedure with respect to the preferential semantics. Our conjecture is that this

procedure is complete based on its principle and the completeness of a proof procedure designed by Ross in [Ros92]. The main difficulty to prove the completeness of our proof procedure comes from the lack of a constructive definition for a preferential extension. So we should reformalize the definition of an extension in the preferential semantics by finding a new constructive method. The proof method used by Ross in [Ros92] has provided us a very good guidance for our further research.

Another research topic about proof theory for preferential semantics could be the computational complexity. If we cannot prove our problem (finding a query in all the extensions of a logic program) to be NP-complete, we may be able to make our proof procedure tractable. This will force us to find new methods for computing cross consistency checks, or replace cross consistency checks with other strategies which are tractable.

# Bibliography


[AE82] K. Apt and M. Van Emden. Contributions to the theory of logic programming. *Journal of ACM*, 29(3):841—862, 1982.

[Bry89] F. Bry. Logic programming as constructivism: A formalization and its application to databases. In *Proceedings of the 8th ACM Symposium on principles of Tatbase systems*, pages 34—50, 1989.

[Cla78] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293—322. Plenum Press, 1978.

[DG84] W. Dowling and J. Gallier. Linear algorithms for testing the satisfiability of propositional horn formula. *Journal of Logic Programming*, 3:267—284, 1984.

[Dun91] P. Dung. Negations as hypotheses: An abductive foundation for logic programming. In *Logic Programming: Proceedings of the 8th International Conference*, pages 3—17, 1991.

[EK76] M. Van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of ACM*, 23(4):37—54, 1976.

[EK89] K. Eshghi and R. Kowalski. Abduction compared with negation by failure. In *Proceedings of the 6th International Conference on Logic Programming*, pages 234—254, 1989.

[Fit85] M. Fitting. A kripke-kleene semantics for logic programs. *Journal of Logic Programming*, 2:295—312, 1985.

[Gel92] A. Van Gelder. The alternative fixpoint of logic program with negation. *Journal of Computer system science*, 29:123—134, 1992.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conf./Sympo. on Logic Programming*, pages 1071—1080, 1988.

[GMS93] L. Giordano, A. Matelli, and M. Sapino. A semantics for eshghi and kowalski's abductive procedure. In *Proceedings of the 10th international conference on logic programming (to appear)*, 1993.

[GN87] M. Genesereth and N. Nilsson. *Logical Foundations Artificial Intelligence*. Morgan Kaufmann Publishers Inc., 1987.

[Goe92] R. Goeble. Xtheorists. XTheorists Manual, 1992.

[GRS91] A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 99(99):620—650, 1991.

[JLM86] J. Jaffer, J-L. Lassez, and M.J.Maher. Some issues and trends in the semantics of logic programming. In *Proceedings of 3rd Inernational Conference on Logic Programming*, pages 223—241, 1986.

[Kow79] R. Kowalski. *Logic for problem solving*. North Holland, 1979.

[KS91] H. Kautz and B. Selman. Hard problems for simple default logics. *Artificial Intelligence*, 49:243—279, 1991.

[Llo87] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[LM84] J. Lassez and M. Maher. Closures and fairness in the semantics of programming logic. *Theoretical Computer Science*, 29:167—184, 1984.

[Moo85] R. Moore. Semantical consideration on nonmonotonic logic. *Artificial Intelligence*, 25(1):234—252, 1985.

[Poo88] D. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36:27—47, August 1988.

[PP93] H. Przymusinska and T. Przymusinski. Semantics issues in deductive databases and logic programming. In A. Banjeri, editor, *Sourcebook on the formal approaches in artificial intelligence*, 1993.

[Prz88] T. Przymusinski. On the decarative semantics of deductive databases and logic programs. In H. Gallaire and J. Minker, editors, *Foundations of deductive databases and Logic programming*, pages 193—216. Morgan Kaufmann Publishers, Inc., 1988.

[Prz89a] T. Przymusinski. Every logic program has a natural stratification and an iterated fixed point model. In *Proceedings of the 8th ACM Symposium on principles of Tatbase systems*, pages 22–33, 1989.

[Prz89b] T. Przymusinski. On the decarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5:167—205, 1989.

[Rei78] R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55—76. Plenum Press, 1978.

[Rob65] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of ACM*, 12(1):23—41, 1965.

[Ros92] K. Ross. A procedural semantics for well-founded negation in logic programs. *Journal of Logic Programming*, 13:1—22, 1992.

[RT88] K. Ross and T. Topor. Inferring negative information from disjunctive databases. *Journal of Automatic Reasoning*, 4:397—424, 1988.

[Tar77] S. Tarnlund. Horn clause compatibility. *BIT*, 17:215—226, 1977.

[YLY93] J. You, L. Li, and L. Yuan. Construction of preferential answer sets for logic programs and default theories. In *Proceedings of the first international workshop on deductive database (to appear)*, 1993.